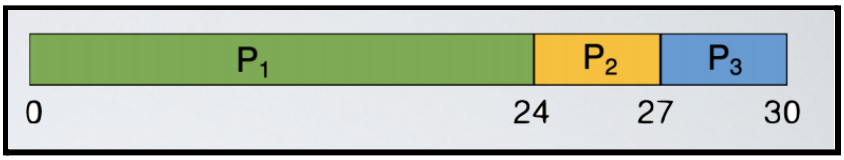
Lecture Notes:

- **The scheduling problem:**
- Suppose we have n threads ready to run and k CPUs, where k ≥ 1. Which jobs should we assign to which CPU(s) and for how long?
- Furthermore, we don't want **starvation** to occur.
- **Starvation** is when a thread is prevented from making progress because some other thread has the resource it requires (could be CPU or a lock).
- Starvation is usually a side effect of the scheduling algorithm.
  E.g. A high priority thread always prevents a low priority thread from running.
- Starvation can also be a side effect of synchronization.
- **Scheduling Criteria:**
- There are 3 main scheduling criterias:
  1. **Throughput:** This is the number of threads that complete per unit time.
     I.e. It is the number of jobs/time (Higher is better)
  2. **Turnaround time:** This is the time for each thread to complete.
     I.e. It is Time_finish – Time_start (Lower is better)
  3. **Response time:** This is the time from request to first response.
     I.e. It is the time between waiting to ready transition and ready to running transition.
     Time_response – Time_request (Lower is better)
- The above criterias are affected by secondary criteria. There are 2 secondary criterias:
  1. **CPU utilization:** This is the percent of time the CPU is doing productive work.
  2. **Waiting time:** This is the average time each thread waits in the ready queue.
- **How to balance criteria:**
- **Batch systems** (supercomputers) strive for job throughput and turnaround time.
- **Interactive systems** (personal computers) strive to minimize response time for interactive jobs. However, in practice, users prefer predictable response time over faster but highly variable response time. Often, personal computers are optimized for an average response time.
- **Two kinds of scheduling algorithms:**
  1. **Non-preemptive scheduling:**
  - This is good for batch systems.
  - Once the CPU has been allocated to a thread, it keeps the CPU until it terminates.
  - Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time.
  2. **Preemptive scheduling:**
  - This is good for interactive systems.
  - The CPU can be taken from a running thread and allocated to another.
  - Preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

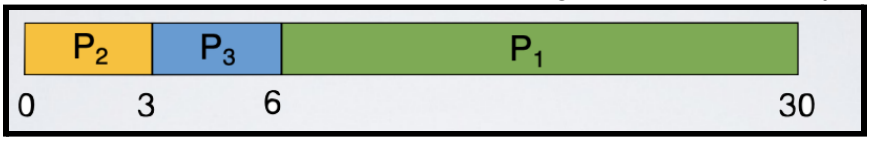
- **FCFS - First Come First Serve:**
- A type of non-preemptive scheduling.
- Here, jobs are run in the order that they arrive and there are no interrupts.
- Consider this example:
  We have 3 threads, P1, P2 and P3 that all start at time 0 and go in that order.
  Suppose P1 takes 24 seconds, P2 takes 3 seconds and P3 takes 3 seconds.



Here are the 3 main criterias for this example:

Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(24 + 27 + 30) / 3 = 27 sec in average
Waiting Time	(0 + 24 + 27) / 3 = 17 sec in average

- From this example, we can see a huge problem with this method. All other threads wait for the one big thread to release the CPU. This is known as the **Convoy Effect**.
- **SJF - Shortest-Job-First:**
- A type of non-preemptive scheduling.
- Here, jobs are run based on their processing time in order from least to greatest.
  I.e. Choose the thread with the shortest processing time.
- Consider this example:
  We have 3 threads, P1, P2 and P3 that all start at time 0.
  Suppose P1 takes 24 seconds, P2 takes 3 seconds and P3 takes 3 seconds.
  Since P2 takes the least amount of time, it goes first, followed by P3 and then P1.



Here are the 3 main criterias for this example:

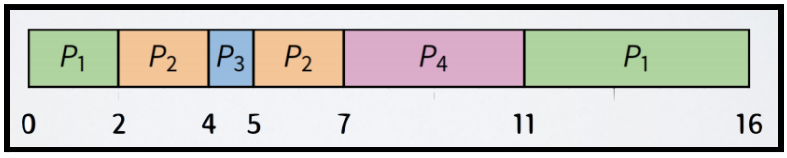
Throughput	3 / 30 = 0.1 jobs/sec
Turnaround	(30 + 3 + 6) / 3 = 13 sec in average
Waiting Time	(0 + 3 + 6) / 3 = 3 sec in average

- The problem with this method is that we need to know processing time in advance.
- Easy to implement in batch systems where the required CPU time is known in advance.
- Impossible to implement in interactive systems where the required CPU time is not known.

- **SRTF - Shortest-Remaining-Time-First:**
- A type of preemptive scheduling.
- In this approach, if a new thread arrives with a CPU burst length less than the remaining time of the current executing thread, then preempt the current thread.
- Consider this example:
  We have 5 threads, P1, P2, P3, P4 and P5 with the following information

Process	Arrival Time	Burst Time
P_1	0	7
P_2	2	4
P_3	4	1
P_4	5	4

This is the order each process runs and finishes.



P1 runs from time 0 to 2. However, at time 2, P2 arrives and since its burst time (4) is smaller than P1's remaining time (5), P2 now runs.

P2 runs from time 2 to 4. However, at time 4, P3 arrives and since its burst time (1) is smaller than P2's remaining time (2) and P1's remaining time (5), P3 now runs.

P3 runs from time 4 to 5 and finishes. At time 5, P4 arrives. Right now, P2 has the shortest time (2), compared to P4 (4) and P1 (5), so P2 runs.

P2 runs from time 5 to 7 and finishes. Now, we have 2 processes P1 and P4. P4 has a time of 4 while P1 has a time of 5, so P4 runs.

P4 runs from time 7 to 11 and finishes. Since P1 is the only process left, it runs from time 11 to 16 and finishes.

- The advantage of this method is that it optimizes waiting time.
- The problem with this method is that it can lead to starvation. In our example above, even though P1 was the first process created, it was the last to finish. Small processes can starve larger processes.
- **RR - Round Robin:**
- A type of preemptive scheduling.
- Each job is given a time slice called a **quantum**, and we preempt each job after the duration of its quantum, moving it to the back of the FIFO queue.
- The advantage of this method is that it has a fair allocation of CPU and a low waiting time (interactive).
- The problem with this method is that there is no priority between threads.
- Context switching is used to save states of preempted processes. Since context switches are frequent and need to be very fast, we want the quantum to be much larger than the context switch cost. The majority of bursts should be less than the quantum but you don't want the quantum to be so big that the system reverts to FCFS.
- Typical values for a quantum is between 1–100 ms.

- **MLQ - Multilevel Queue Scheduling:**
- A type of preemptive scheduling.
- Each thread is associated with a priority and the highest priority thread(s) are executed before lower priority thread(s). If multiple threads have the same priority, then do round-robin.
- E.g.

Here, we have 3 threads (T1, T3, T6) that are high priority, 1 thread (T4) that is medium priority and 2 threads (T2 and T5) that are low priority.

The high-priority threads will be executed first, and since there are multiple high-priority threads, round-robin will be used.

After all the high-priority threads have been executed, the medium-priority threads will be executed. Since there is only 1, there is no need to do round-robin.

After all the medium-priority threads have been executed, the low-priority threads will be executed, and since there are multiple low-priority threads, round-robin will be used.

- Some problems with this method:
    1. Starvation of low priority thread(s).
    2. Possible starvation of high priority thread(s).
    3. Priorities are arbitrarily decided. How do we decide on the priority?
- Example of starvation of high priority threads:
    - Suppose we have 3 threads, T1 (low priority), T2 (medium priority) and T3 (high priority) and a lock, L.
    - T1 starts, runs and acquires L.
    - T2 starts, preempts the CPU and runs.
    - T3 starts, preempts the CPU, runs but gets blocked while trying to acquire L.
    - T2 is elected to run as it is the highest priority thread to be ready to run.

    - As you can see, T3 (the high priority thread) is now starved.
    - A solution to this issue is **priority donation**.
- Example of priority donation:
    - Suppose we have 3 threads, T1 (low priority), T2 (medium priority) and T3 (high priority) and a lock, L.
    - T1 starts, runs and acquires L.
    - T2 starts, preempts the CPU and runs.
    - T3 starts, preempts the CPU, runs but gets blocked while trying to acquire L.
    - T3 gives its high priority to T1.
    - T1 (now high priority) runs, releases the lock and returns to low priority immediately after.
    - T3 (now unblocked) preempts the CPU and runs.

- To prevent starvation of low priority threads, change the priority over time by either increase the priority as a function of waiting time or decrease the priority as a function of CPU consumption.
- To decide on the priority, observe and keep track of the thread CPU usage.
- **MLFQ - Multilevel Feedback Queue Scheduling:**
- A type of preemptive scheduling.
- This is the same as MLQ but it changes the priority of the process based on observations.
- This is a Turing-award winner algorithm.
- Observations:

Rule 1	If Priority(A) > Priority(B), A runs
Rule 2	If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue
Rule 3	When a job enters the system, it is placed at the highest priority (the topmost queue)
Rule 4	Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue)
Rule 5	After some time period S, move all the jobs in the system to the topmost queue

Textbook Notes:

- **Mechanism - Limited Direct Execution:**
- Introduction:
- In order to virtualize the CPU, the operating system needs to somehow share the physical CPU among many jobs running seemingly at the same time. The basic idea is simple: run one process for a little while, then run another one, and so forth. By time sharing the CPU in this manner, virtualization is achieved.
- There are a few challenges, however, in building such virtualization machinery.
    1. Performance. How can we implement virtualization without adding excessive overhead to the system?
    2. Control. How can we run processes efficiently while retaining control over the CPU? Control is particularly important to the OS, as it is in charge of resources; without control, a process could simply run forever and take over the machine, or access information that it should not be allowed to access. Obtaining high performance while maintaining control is thus one of the central challenges in building an operating system.
- Basic Technique - Limited Direct Execution:
- **Direct execution** simply means just run the program directly on the CPU. Thus, when the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry point, jumps to it, and starts running the user's code.
- This approach gives rise to a few problems:
    1. If we just run a program, how can the OS make sure the program doesn't do anything that we don't want it to do, while still running it efficiently?
    2. When we are running a process, how does the operating system stop it from running and switch to another process, thus implementing the time sharing we require to virtualize the CPU?
- Problem #1 - Restricted Operations:
- Direct execution has the obvious advantage of being fast; the program runs natively on the hardware CPU and thus executes as quickly as one would expect. But running on the CPU introduces a problem: what if the process wishes to perform some kind of restricted operation, such as issuing an I/O request to a disk, or gaining access to more system resources such as CPU or memory?
- A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.
- One approach would simply be to let any process do whatever it wants in terms of I/O and other related operations. However, doing so would prevent the construction of many kinds of systems that are desirable. For example, if we wish to build a file system that checks permissions before granting access to a file, we can't simply let any user process issue I/Os to the disk; if we did, a process could simply read or write the entire disk and thus all protections would be lost.
- Thus, the approach we take is to introduce a new processor mode, known as **user mode**. Code that runs in user mode is restricted in what it can do. For example, when running in user mode, a process can't issue I/O requests; doing so would result in the processor raising an exception; the OS would then likely kill the process.
- In contrast to user mode is **kernel mode**, which the operating system or kernel runs in. In this mode, code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.
- To enable user processes to perform some kind of privileged operation, virtually all modern hardware provides the ability for user programs to perform a **system call**.

- System calls allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocating more memory.
- To execute a system call, a program must execute a special **trap instruction**. This instruction simultaneously jumps into the kernel and raises the privilege level to kernel mode. Once in the kernel, the system can now perform whatever privileged operations are needed (if allowed), and thus do the required work for the calling process. When finished, the OS calls a special **return-from-trap instruction**, which returns into the calling user program while simultaneously reducing the privilege level back to user mode.
- The hardware assists the OS by providing different modes of execution. In user mode, applications do not have full access to hardware resources. In kernel mode, the OS has access to the full resources of the machine. Special instructions to trap into the kernel and return-from-trap back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the **trap table** resides in memory.
- The hardware needs to be a bit careful when executing a trap, in that it must make sure to save enough of the caller's registers in order to be able to return correctly when the OS issues the return-from-trap instruction. On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**. The return-from trap will pop these values off the stack and resume execution of the usermode program.
- The kernel must carefully control what code executes upon a trap. The user/user mode can't arbitrarily jump into a memory address in the kernel. The kernel does so by setting up a trap table at boot time. When the machine boots up, it does so in kernel mode, and thus is free to configure machine hardware as needed. One of the first things the OS does is to tell the hardware what code to run when certain exceptional events occur. The OS informs the hardware of the locations of these **trap handlers**, usually with some kind of special instruction. Once the hardware is informed, it remembers the location of these handlers until the machine is next rebooted, and thus the hardware knows what to do (i.e., what code to jump to) when system calls and other exceptional events take place.
- To specify the exact system call, a **system-call number** is usually assigned to each system call. The user code is thus responsible for placing the desired system-call number in a register or at a specified location on the stack. The OS, when handling the system call inside the **trap handler**, examines this number, ensures it is valid, and, if it is, executes the corresponding code. This level of indirection serves as a form of protection; user code cannot specify an exact address to jump to, but rather must request a particular service via number.
- Problem #2 - Switching Between Processes:
- The next problem with direct execution is achieving a switch between processes.
- If a program is running on the CPU, that means the OS isn't running. We need to devise a solution for the operating system to regain control of the CPU so that it can switch between processes.
- One approach that some systems have taken in the past is known as the **cooperative approach**. In this style, the OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.
- Most processes transfer control of the CPU to the OS quite frequently by making system calls. Systems like this often include an explicit yield system call, which does nothing except to transfer control to the OS so it can run other processes.

- Applications also transfer control to the OS when they do something illegal. For example, if an application divides by zero, or tries to access memory that it shouldn't be able to access, it will generate a trap to the OS. The OS will then have control of the CPU again and likely terminate the offending process.
- While the cooperative approach sounds good in theory, in practice, it is terrible. Malicious software/programs can take control of the CPU forever.
- A **non-cooperative approach** would be to use a timer. A **timer device** can be programmed to raise an interrupt every so many milliseconds. When the interrupt is raised, the currently running process is halted, and a pre-configured interrupt handler in the OS runs. At this point, the OS has regained control of the CPU.
- The OS must inform the hardware of which code to run when the timer interrupt occurs, which it does, at boot time. Also during the boot sequence, the OS must start the timer.
- Saving and Restoring Context:
- Now that the OS has regained control, whether cooperatively via a system call, or more forcefully via a timer interrupt, a decision has to be made: whether to continue running the currently-running process, or switch to a different one. This decision is made by a part of the operating system known as the **scheduler**.
- If the decision is made to switch, the OS then executes a **context switch**. A context switch is conceptually simple: all the OS has to do is save a few register values for the currently-executing process and restore a few for the soon-to-be-executing process. By doing so, the OS thus ensures that when the return-from-trap instruction is finally executed, instead of returning to the process that was running, the system resumes execution of another process.
- To save the context of the currently-running process, the OS will execute some low-level assembly code to save the general purpose registers, PC, and the kernel stack pointer of the currently-running process, and then restore said registers, PC, and switch to the kernel stack for the soon-to-be-executing process. By switching stacks, the kernel enters the call to the switch code in the context of one process (the one that was interrupted) and returns in the context of another (the soon-to-be-executing one). When the OS then finally executes a return-from-trap instruction, the soon-to-be-executing process becomes the currently-running process. And thus the context switch is complete.
- **Scheduling - Introduction:**
- Workload Assumptions:
- We will make the following assumptions about the processes, sometimes called jobs, that are running in the system:
    1. Each job runs for the same amount of time.
    2. All jobs arrive at the same time.
    3. Once started, each job runs to completion.
    4. All jobs only use the CPU (I.e. They perform no I/O).
    5. The run-time of each job is known.
- Scheduling Metrics:
- We also need a **scheduling metric**. For now, our scheduling metric will be the turnaround time.
- The **turnaround time** of a job is defined as the time at which the job completes minus the time at which the job arrived in the system.
  $T_{turnaround} = T_{completion} - T_{arrival}$
- Because we have assumed, for now, that all jobs arrive at the same time, $T_{arrival} = 0$ and hence $T_{turnaround} = T_{completion}$.
- You should note that turnaround time is a performance metric.

- First In, First Out (FIFO):
- The most basic algorithm we can implement is known as **First In, First Out (FIFO) scheduling** or sometimes **First Come, First Served (FCFS)**.
- FIFO has a number of positive properties: it is clearly simple and thus easy to implement.
- However, the main problem with this approach is the **convoy effect**. This occurs when a number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.
- Shortest Job First (SJF):
- In this approach, the shortest job is run first, then the next shortest, and so on.
- However, SJF also has a problem with the convoy effect.
- E.g. Suppose we have 3 processes, A, B and C. Suppose A arrives at t = 0 and runs for 100 seconds, B arrives at t = 10 and runs for 10 seconds and C arrives at t = 20 and runs for 10 seconds. Both B and C are blocked from running, even though they have a shorter run time than A, because they arrived later.
- Shortest Time-to-Completion First (STCF):
- To address this concern, we need to relax assumption 3 (that jobs must run to completion).
- In this approach, if a new thread arrives with a CPU burst length less than the remaining time of the current executing thread, then preempt current thread.
- A New Metric - Response Time:
- We define **response time** as the time from when the job arrives in a system to the first time it is scheduled.
- T_response = T_firstrun − T_arrival
- None of the 3 above methods (FIFO, SJF and STCF) are particularly good for response time.
- Round Robin:
- To solve this problem, we will introduce a new scheduling algorithm, classically referred to as **Round-Robin (RR)**.
- The basic idea is simple: instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished.
- Note that the length of a time slice must be a multiple of the timer-interrupt period.
- The length of the time slice is critical for RR. The shorter it is, the better the performance of RR under the response-time metric. However, making the time slice too short is problematic: suddenly the cost of context switching will dominate overall performance. Thus, deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to amortize the cost of switching without making it so long that the system is no longer responsive.
- Note that the cost of context switching does not arise solely from the OS actions of saving and restoring a few registers. When programs run, they build up a great deal of state in CPU caches, TLBs, branch predictors, and other on-chip hardware. Switching to another job causes this state to be flushed and a new state relevant to the currently-running job to be brought in, which may exact a noticeable performance cost.
- RR is good with response time but bad with turnaround time.

- **Scheduling - The Multi-Level Feedback Queue (MLFQ):**
- The fundamental problem MLFQ tries to address is two-fold. First, it would like to optimize turnaround time. Second, MLFQ would like to make a system feel responsive to interactive users and thus minimize response time
- MLFQ - Basic Rules:
- **Rule 1:** If Priority(A) > Priority(B), A runs & B doesn't.
- **Rule 2:** If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue).
- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- **Rule 5:** After some time period S, move all the jobs in the system to the topmost queue.
- In our treatment, the MLFQ has a number of distinct queues, each assigned a different priority level. At any given time, a job that is ready to run is on a single queue. MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (I.e. A job on a higher queue) is chosen to run.
- Of course, more than one job may be on a given queue, and thus have the same priority. In this case, we will just use round-robin scheduling among those jobs.
- The key to MLFQ scheduling therefore lies in how the scheduler sets priorities. Rather than giving a fixed priority to each job, MLFQ varies the priority of a job based on its observed behavior. If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority. In this way, MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior.